Evolution of a common controller

D. Powell[*], D. Barbour, G. Gilbreath
Space and Naval Warfare Systems Center Pacific
53560 Hull Street, San Diego, CA 92152

ABSTRACT

Precedent has shown common controllers must strike a balance between the desire for an integrated user interface design by human factors engineers and support of project-specific data requirements. A common user-interface requires the project-specific data to conform to an internal representation, but project-specific customization is impeded by the implicit rules introduced by the internal data representation. Space and Naval Warfare Systems Center Pacific (SSC Pacific) developed the latest version of the Multi-robot Operator Control Unit (MOCU) to address interoperability, standardization, and customization issues by using a modular, extensible, and flexible architecture built upon a shared-world model. MOCU version 3 provides an open and extensible operator-control interface that allows additional functionality to be seamlessly added with software modules while providing the means to fully integrate the information into a layered game-like user interface. MOCU's design allows it to completely decouple the human interface from the core management modules, while still enabling modules to render overlapping regions of the screen without interference or a priori knowledge of other display elements, thus allowing more flexibility in project-specific customization.

Keywords: common, unmanned, user interface, robot, control, MOCU

1. INTRODUCTION

Military air, ground, and surface unmanned systems continue to require a human in the control loop to meet safety concerns and physical environment constraints. Each mission and its associated unmanned systems bring unique human-machine interface requirements, but there exists a common cross-section of functionality that becomes the basis for the concept of a common controller.

Many "common" controllers are being developed by both the Government and industry with the lure of reduced development and training costs by standardization and reuse of display and backend components. "Common" is used to describe the ability of a single controller to communicate to multiple systems. The term's meaning varies to specify control of multiple systems speaking a protocol standard, providing similar capabilities, or across domains. At SSC Pacific, our goal is not to create one common controller with limited control of each system, but to define a *common software framework* built upon modularity and flexibility to enable control of any vehicle or sensor through a multimodal interface.

2. BACKGROUND

MOCU was originally conceived in 2001 to reduce the time and effort expended on custom Operator Control Units (OCUs) for each project that were largely duplicative of existing OCUs. The unique elements of each application were often limited to communication protocols and variations on how data was displayed to the user. These problems were solved in the first version of MOCU by supporting dynamically loaded modules, each following a specified Application Programming Interface (API) based on its purpose or 'type' (vehicle-control protocol, map display, video transport, and so on) [1].

Maintaining multiple APIs quickly became onerous and did not adapt efficiently to requirements for new payloads and other module types. In addition, each module communicated only with the 'core' OCU component and could not effectively share information with or adapt to the behavior of other modules.

MOCU 2 solved these problems by introducing a new flexible API able to handle the requirements of existing modules and capable of supporting efficient inter-module communication. The API uses a shared-world model accessed by a

[*] darren.powell@navy.mil; phone 1 619 553-2553; www.public.navy.mil/spawar/Pacific/Robotics

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **APR 2012**	2. REPORT TYPE	3. DATES COVERED **00-00-2012 to 00-00-2012**

4. TITLE AND SUBTITLE
Evolution of a common controller

5a. CONTRACT NUMBER
5b. GRANT NUMBER
5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

5d. PROJECT NUMBER
5e. TASK NUMBER
5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Space and Naval Warfare Systems Center Pacific,53560 Hull Street,San Diego,CA,92152

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Precedent has shown common controllers must strike a balance between the desire for an integrated user interface design by human factors engineers and support of project-specific data requirements. A common user-interface requires the project-specific data to conform to an internal representation, but project-specific customization is impeded by the implicit rules introduced by the internal data representation. Space and Naval Warfare Systems Center Pacific (SSC Pacific) developed the latest version of the Multi-robot Operator Control Unit (MOCU) to address interoperability standardization, and customization issues by using a modular, extensible, and flexible architecture built upon a sharedworld model. MOCU version 3 provides an open and extensible operator-control interface that allows additional functionality to be seamlessly added with software modules while providing the means to fully integrate the information into a layered game-like user interface. MOCU's design allows it to completely decouple the human interface from the core management modules, while still enabling modules to render overlapping regions of the screen without interference or a priori knowledge of other display elements, thus allowing more flexibility in project-specific customization.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**9**	

publish/subscribe messaging paradigm. Modules now conform to a runtime-extensible data model that extends to include additional describing attributes without adversely affecting other modules. Similarly a module with limited capability may publish a subset of the information. But the user interface was hardwired to read certain information from the shared-world model and only had a limited capability of displaying extended attributes on the display.

Both MOCU 1 and MOCU 2 provided a relatively flexible user interface (compared to other OCUs), but still prevented users and developers from making desired modifications. For example, the MOCU core defined the behavior of mouse clicks and movements and this behavior could not be changed without modifying the core software. This shortcoming is addressed in MOCU 3 by splitting the core code into modules as well. All user interactions and displays are contained in the shared-world model and any the core modules can be replaced or selectively chosen to provide the functionality required for a particular project.

3. DESIGN

3.1 Design criteria

Many factors shaped our architectural and implementation decisions. Key among these are:

1. *Configuration*: The display and operation of MOCU must be configurable via text files. For gauges and configurable text displays, this requires the ability to identify the data sources, and support mathematical expressions. For buttons, joysticks, and menus, this requires the ability to script commands or modify the appropriate control data. The ability to write these scripts implies the data will be available in predictable structures with consistent identifiers.
2. *System Discovery*: MOCU dynamically discovers robots via the protocol modules. Supporting system discovery implies that modules may not have access to data they need, especially startup.
3. *Integrated Display*: The user interface must reflect the modularity of the system but still allow the layout to fuse and overlay data to maximize screen real estate and operator effectiveness.
4. *Control Feedback*: When a control succeeds or fails, operators need to receive useful feedback.
5. *Streaming Data*: One form of streaming is video data provided by a protocol from a camera. This data has unique requirements when routing through the system. For instance, video may be passed to a codec module that can decipher the video stream which is then forwarded to one or more listeners including display windows.
6. *Avoid Polling*: Event-driven designs prevent polling, which can consume valuable CPU cycles. For instance, continuously polling a large route for changes is inefficient and would result in poor performance and unacceptable latency.
7. *Delta Isolation*: Support for efficient delta-isolation requires knowing which changes occur so that one may process only the changes. For example, it is necessary to deliver just the changes to the robot rather than issuing a new mission when an active route or mission is modified.
8. *Performance*: Low-latency access and mutation of the data reduces need to perform module-local caching and similar complexities. Most OCU operations benefit, such as loading project-configured context menus or resizing routes.

3.2 Architecture design

The configuration and system-discovery requirements infer that data about the world, the mission, and the OCU must be presented in a predictable manner. This implies the use of a shared-world model with which each module can interact. A shared-world model consists of a shared database representing a predefined ‘view’ of the world. In an OCU, the model includes mission plans, routes, the display and configuration of the OCU itself, robots, zones, tactical objects, and even the operator. Modules keep the model up-to-date, continuously feeding in any changes during runtime, but configuration data can be persisted in files and loaded directly into the model at startup.

An integrated display is achieved by providing a layered window framework contained within the operating system (OS) windows. Each module is able to draw to its specified windows orchestrated by a window-manager module to achieve the requested layering. Hardware acceleration using OpenGL greatly reduces the cost of seamlessly overlapping the

windows. The result enables the creation of game-like interfaces where the modules and layers of the system are hidden from the end-user.

The requirements for delta isolation and to avoid polling imply that one can subscribe to specific changes in the shared-world model. These requirements drive the decision towards publish/subscribe. In a pure shared-world model, commands are issued by manipulation of active goals, such as manipulating 'goal rudder' and 'goal throttle' properties in order to control an unmanned surface vehicle (USV) by joystick. These properties reflect the operator's goals, and it becomes the job of the protocol and control modules to turn the operator's wishes into commands. Unfortunately, this purist approach is insufficient to meet the control feedback requirement: it won't always be the case that the protocol module is in a state where it can obey commands issued in this manner, and there are no obvious channels to provide such feedback. The architecture must provide another mechanism for control.

One potential solution, favorable because it doesn't involve any additional API features, is to implement a miniature blackboard system sometimes used in artificial intelligence. In a blackboard system commands and their results are represented as objects in shared spaces called blackboards. [2] Modules subscribe to blackboard objects and examine command objects, searching for those they can execute. When the module discovers such an object, it is removed from the blackboard and executed, issuing commands as necessary and adding a results object when complete. To issue an order, a module creates a command object. But this blackboard system approach does have an unfortunate characteristic: there is no clear means to determine whether a command will be executed or sit indefinitely on the blackboard.

To solve the issue of command delivery with feedback, we decided to support the association of methods with objects in the shared-world model. In the API, modules can easily test which methods are associated with an object, receive feedback for whether a call was accepted, and even associate a callback with each method call.

Of the above design criteria, only support for streaming data remains. In a shared-world model a data stream can be represented as either sequential updates to identifiable model states or as a collection of message objects, but both solutions are problematic. For sequential updates, it is impossible to distinguish two messages with identical content that result in no update, and it would be easy to miss messages that are often critical to interpreting the stream, as with encoded video. For message objects, no module would know when every other module is done observing the message, and thus it would be difficult to delete them. However, the methods introduced for commands will serve also for these data-streaming tasks. In order to support multiple data sinks, such as multiple windows displaying the same video, one creates a listener object to which modules subscribe by associating with it an object containing a receiver method. Messages are placed on the channel simply by issuing a method call with parameters identical to each receiver object.

In summary: 1) The shared-world model supports configuration and system discovery, 2) a seamlessly layered layout enables integrated displays, 3) methods support control feedback and streaming data, and 4) publish/subscribe supports delta isolation and helps avoid polling.

4. IMPLEMENTATION

The module implementing the MOCU publish/subscribe architecture is called the DataServer. This section describes, at a high level, the implementation of this module, breaking it down into the shared-world model, subscriptions, methods, the API, and implementation details.

4.1 Shared-world model

The model is composed of two basic elements: objects and properties.

Objects are organized in a tree structure. Each object has a parent and any number of children. There is no inherent semantic property to the parent-child relationship except that the deletion of a parent causes a cascading delete of all its children. Objects are each identified by a runtime-unique 64-bit integer called a Globally Unique Identifier or GUID (obviously not globally unique but the term has stuck). This GUID is created when an object is initially published to the DataServer. Objects may also be published with a few other attributes including a name, type, and subtype. The name and type information have no semantic meaning to the DataServer. What objects represent is up to agreements between the other modules.

A 'root' object is the ultimate ancestor of all other objects. This supports system discovery because any module can find all objects by starting from the root. Many objects are placed directly under the root, generally including unmanned platforms, routes and other missions, symbols for the map, et cetera.

Each object may have any number of properties, each identified by a unique name. Properties contain simple values, such as integers, Booleans, floating-point numbers, strings, and GUIDs. They can also have meta-data including units, representation information, a simple text description for humans, and range limitations. To allow for complex values, a property value may also contain a list or set of property values.

Manipulations and queries to the model include the basic create, read, update, and delete accesses to individual objects and properties referenced by GUID. To support system discovery, it is also possible to obtain a list of children or properties. Object lists are heavily used in the shared-world model to represent ordering, such as waypoint lists, action lists, and zone-point lists. To support this, special provisions exist for publishing a new object between a parent and child, and for deleting an object and automatically promoting a child in its place.

4.2 Subscriptions

Subscriptions provide modules with the ability to be notified of changes to specific objects or properties within the DataServer. Upon startup, nearly every module makes a subscription and returns to wait for future notifications to wake up and operate on the update. The DataServer API provides two basic subscription options: subscribe to the children of an object or subscribe to the properties of an object. When subscribing to properties, one may filter by name or choose to grab all the properties.

Each subscription includes a function pointer and data pointer that are passed to the DataServer. After an update occurs, a DataServer thread will call back using this function and data pointer. Included will be a reference to the object or properties being updated and a 'notification type' that indicates the nature of the update--usually new, changed, or deleted. When initially subscribing to an object, the associated children and properties are initially received with the new notification, even if other subscriptions within the same module have already been notified. This is critical to the design of new functionality because it allows each instance to make a subscription and be guaranteed an initial value without first polling to get that value.

Most of the time, multiple subscriptions are necessary to arrive at the object or property of interest in the object tree. The list of objects and/or properties that describe the unique location of the object or property is called the *dotted-path*. For instance, if 'root' has a child 'A' and 'A' has a child 'B', the dotted path to 'B' is *root.A.B*. Likewise if 'B' has a property 'c', the path to 'c' would be *root.A.B.c*. Only 'root' is guaranteed at startup, so a subscription must be created for each subsequent object-tree level before arriving at the object or property of interest. To simplify development and reduce implementation errors, helper functions were created to perform these subscriptions.

4.3 Methods

In the model, each method is published with a unique name under an object. At the time of publishing, a method will establish its callback function pointer and data pointer, parameters, callback results, and default call options. Methods are later called with GUID and name, parameters, callback function, callback data pointer, and any call option overrides.

Methods may optionally be queued with an arbitrary delay in milliseconds. Queued methods satisfy the asynchronous property and avoid issues of potentially cyclic calls and long waits before returning after a method call, while immediate-mode method calls support safety for thread-sensitive GUI operations. Delayed method calls are often used by a module to execute a periodic event without creating a thread.

While methods were introduced in order to provide feedback, properties are still often used to issue commands in the purist shared-world-model fashion: representing the operator's goals in a command object. Vehicles have navigation objects where we can set desired heading and speed. Camera objects allow one to set options like iris, shutter, and auto-focus. Pan and tilt objects allow one to set rotation goals on various axes. In order to provide necessary control feedback while still using this shared-world model approach to express goals, we utilize methods to set the operation state. For example, methods will start engines, or request control for remote operation of a vehicle, or assign a particular route to a vehicle. This gives the vehicle the opportunity to refuse or fail and then indicate the reason.

4.4 Scripting

MOCU has long used scripting to support configuration-file changes for user input and gauge definitions. The flexibility to modify behaviors and display components without the need to compile code often outweighs the slight loss in speed of using scripts. For this reason, scripts now make up a large portion of the runtime behaviors and display components including menu definitions, mouse and joystick behaviors, and entire portions of the user interface. HTML/CSS is used for most parts of the user interface that do not require the use of 3D hardware acceleration.

Both the JavaScript and HTML modules extend their languages to include DataServer API calls, making interaction with the DataServer greatly simplified within the scripts. 'Watch' objects, built on top of scripting, further enhance the capability by creating a mechanism which will execute a script at a periodic rate and notify the creator when a criterion is met. The extent of the use of scripts has one downfall in the lack of a development environment. Normal JavaScript or HTML debuggers only partially work due to the extended language used within MOCU.

Although the primary scripting language used within MOCU is JavaScript, the mechanism is included to support multiple and simultaneous scripting modules so long as the scripts are prefixed. For instance all JavaScript scripts are prefixed with '#js:'.

4.5 DataServer implementation details

The implementation details underlying the API are not required to use the DataServer, but they do offer some insight as to its function and how it achieves its desired performance characteristics.

The shared-world model itself is implemented using a large hashtable for all the objects, and each object contains a set of GUIDs representing the children and two hashtables: one for methods and one for properties. All the strings for property names are interned, so the hashes and equality tests simply use their pointer value. This organization provides two-indirection fast lookups for properties and methods even when grabbing a property or method list. State for these objects is represented by reference-counted immutable values, which reduces the need for lock-and-copy operations.

Updates to the shared-world model are performed by the caller thread, but all processing for subscription waits until later, to be performed by a DataServer worker thread, which for each update will locate interested subscribers and process the callbacks. The DataServer runs multiple worker threads that solely perform subscription callbacks and queued or delayed-method calls. Having more than one thread helps reduce concerns about how much time a module is consuming in a worker thread, but does require even greater attention to multi-threaded safety in the modules.

Recently, the DataServer was overhauled to use a single thread per module. It was found the thread pool used in the previous version reduced the resources needed by the OCU hardware but pushed the thread-safety issues into the modules. Each subscription or method callback could be called by any of the DataServer threads requiring modules to design with thread-safety in mind. To alleviate the work of the module designer, the new DataServer can be optionally chosen to guarantee all subscription callbacks are called from one DataServer thread. The new version was also rewritten to use atomic maps internally, reducing the thread-safety requirements and increasing performance by reducing lock collisions. This is an excellent example of the modularity within MOCU.

While there can be tens of thousands of objects and properties representing routes, vehicles, and missions in the shared-world model, there is often a larger set of objects representing subscriptions. During a typical execution of MOCU, the majority of the state in the DataServer is involved with managing these subscriptions, keeping queues of the updates each subscription needs to process, and tracking what that subscription has already seen in order to support delta isolation.

While the API for the DataServer is fairly simple, attempting to achieve high performance, subscription grouping, thread safety, and the variety of other non-functional requirements certainly provides an interesting challenge.

5. PRESENTATION LAYER

Although only one design criterion directly affects the presentation layer, a whole section is devoted to this critical component of MOCU. If the backend API is where the rubber meets the road, the presentation layer is the driver's seat where the operator interfaces with the vehicle. The design, layout, and attention to detail changes the operator's experience from a purely functional and at times unsafe engineer's interface to an intuitive look and feel that is efficient to use.

5.1 Layered layout

The purpose of the presentation layer is to bring multiple sources of information into a unified display. Until version 3, MOCU followed many other OCU strategies in creating a flexible tiled user interface with drag-able dialogs or toolbars. MOCU 3 focuses on enabling integrated and overlapped information in a game-like fashion.

It is natural for a design to gravitate to tiled windows to separate content on an operator's display since the concept is built into all major operating systems. In general, the OS does a nice job of separating content within child windows and

dialogs, effectively making the presentation layer modular and extensible. In addition, a child window can be positioned by configuration files to give the layout flexibility, or dialogs can be created that allow the user to change their location. Multiple windows require either tiling or overlapping, and a tiled layout is often chosen due to concerns about covering critical information.

Strictly tiled displays reduce the amount of data which can be placed into key locations of the screen. The eyes of an operator are physically limited to focusing on a small portion of the screen. If critical data is spread over the screen, the operator is required to scan portions of the screen causing fatigue, or else they will miss an update or alert. Layering the data enables the design to fuse and co-locate information from multiple sources into designated spots on the screen. An example of layering can be found on webpages with popup ads near or on the desired content. The proximity of the ad to the normal page content requires little eye movement to acknowledge the ad in comparison to a flashing ad near the side of the screen which requires peripheral vision to identify.

Display order and transparency are critical to the layering of windows within a display. Modules discover the windows allocated to them by object type in the shared-world model and then publish a method to be called at paint time under the window's object. Modules must be managed to produce the correct and deterministic order. MOCU uses a module called the WindowManager to perform this task. The WindowManager does not display its own content but informs each window when to draw, starting with the bottom-most window.

Much of a display layer can be transparent, often allowing coverage of parts of the screen without blocking previously drawn content. Although OS windows continue to improve in transparency support, we have found creating our own layers within the window to be more efficient. The window system must also understand the mouse input as it relates to order and transparency perform user interactions in a predictable fashion.

Animation is also key to an operator's understanding of the result of their action. The limited resolution for display often requires rearranging the prominence of information on the screen. An example is the display of video and map data, both of which require a large portion of the display depending on the operation being performed by the operator. Switching positions on the screen without animation becomes disorienting to the operator if animation does not show the exchange of positions.

Figure 1. This display configuration allows one of three windows to be displayed in the larger location. The windows are animated between locations to show the user which screens are exchanging position. This example shows three screenshots while the user chooses an enlarged drive camera in place of a payload camera.

5.2 Tailor to the warfighter

Layering allows the user interface to be tailored to its audience, namely the 20-year-old warfighter. A key inspiration for two MOCU layouts came from analyzing the latest concepts in game development, specifically how information was displayed to the gamer. A standard first-person shooter (FPS) immerses its player into the game's map with concise information overlaid on the screen as feedback to the avatar's status. The average teenager is able to grab a joystick in one of these games and start navigating the scene with little training. There are few menus or other need for mouse clicks and the gamer is able to make quick decisions and adjustments to the navigation of their avatar.

Tactical control of robots is very similar to an FPS game. The operator is heavily focused on sensor feedback, often video, and only needs a relative map for normal operation. The operator requires the ability to make quick decisions and modify behavior based on sensor feedback. Immersion into the robot's view not only reduces training but also enables more efficient control of the unmanned system.

5.3 Minimize barriers

MOCU enables multiple modules to draw to the screen to allow for future extensibility. Visualization of robotic sensors and data feeds could not be handled within MOCU 2. The core software acted as a barrier in the system that required updates when new types of data or interfaces were needed between a module and the user interface. It could not be adequately extended to handle all situations regarding menus, 3D-display elements, or general windowing. MOCU now puts every module on equal footing and enables more transparency between the user interface and the modules, thus removing the barrier effect found in older versions. Much of the precompiled code within the core was replaced with JavaScript, which can be modified based on the configuration.

6. LIMITATIONS

6.1 Performance versus distribution

MOCU is designed to be near-real-time while still maintaining an enormous amount of flexibility. By loading modules and maintaining its shared-world model in its memory at runtime, the in-memory DataServer guarantees order of events, timing, and reliability.

A distributed system is highly desirable for multiple reasons. If designed correctly, it allows for computational load balancing across hardware, multiple operators, and redundancy. But it comes at a cost——even 'deterministic' or 'near real-time' protocols require communication over a network with associated packet loss and latency.

MOCU was originally designed to work at a tactical level with one operator controlling multiple vehicles and sensors, but it has also operated within an enterprise solution as the presentation component. In these systems, one or more modules are created to interface to the system's service bus. MOCU still enables the flexible user interface with the ability to support new interfaces by adding modules. MOCU has also acted as a headless adapter translating from an external interface to the service bus.

6.2 Ad-hoc queries and subscriptions

The current API for queries and subscriptions is quite limited. There are no joins, native dotted-path queries, or subscribing to functional reactive expressions. Instead, the shared-world model is queried procedurally, navigating links between objects and grabbing properties and objects by lists. The limited expressiveness of the API greatly simplifies the implementation of the DataServer. However, the cost is an increase in complexity of many modules that use this API.

To address the complexity faced by the developer, we have implemented an extensive library for common tasks such as grabbing or subscribing to a collection of objects associated with a dotted path. These helper classes are built upon the DataServer API to isolate the API from a design-specific implementation; however, this comes at an efficiency cost. A more robust API would allow for optimizations to be made within the DataServer.

6.3 Shared-world model versus flexibility

By nature, any shared model tends to become rigid and fragile as components and configuration data are developed to interact with it. Changing a mature model might require updates to many modules and deprecation of configuration data. Achieving flexibility under these conditions takes foresight, good requirements analysis, and up-front design. This flexibility, in turn, has its own costs in terms of complexity and performance for the modules that implement or interact with the associated features.

While this is a limitation of our architecture and of shared-world models, we can honestly claim that we are still in a better situation than we were while maintaining several task-specific APIs. Modifications to existing functionality are similar to the compatibility maintenance issues found with the task-specific APIs; however, the shared-world model allows its current model to be extended without affecting existing listeners. For example, a route object with a 'color' property describing its presentation may represent an RGB color. If a new display module can use the alpha channel, a new property could be added to the route object called 'alpha' without any effect to the existing modules that understand and use the 'color' property.

In order to ensure MOCU can be flexibly adapted to user requirements, we do our best to apply our foresight. But use of a shared-world model in a world where user requirements change with new platforms, new payloads, and new operator input devices, and new features guarantees that updates and modifications are inevitable.

7. CURRENT AND FUTURE WORK

MOCU is used in multiple DoD programs of record, academia, and other research work. The following lists a few of the active projects.

The Man Transportable Robotic System (MTRS) program provided a solution to an immediate need in the explosive ordnance disposal (EOD) community. Variations of the QinetiQ *Talon* and iRobot *PackBot* were fielded, each with its own OCU and each with proprietary interfaces to remotely control the vehicles. The result demonstrated the need for common-control interfaces and common-user-interface software. MOCU has demonstrated control of these two systems in support of a Joint Urgent Operational Needs Statement (JUONS). In addition, the Advanced EOD Robotic System (AEODRS) aims to solve these issues by standardizing the interface to the vehicles while also standardizing on MOCU as the common software.

Figure 2. An early user interface design for AEODRS. The display configuration is designed for a small handheld OCU to control a small tracked vehicle with a manipulator. The small windows on the left can be swapped into the main area on the right to enable a larger view of the content. The tiled display with temporary overlays was chosen to minimize content obstruction.

The mission packages on the Littoral Combat Ship (LCS) will operate air, surface, and underwater vehicles as part of their planned missions. The current design requires separate control software for each of these systems. This has prompted the push for an open architecture that will combine the common elements of these control systems. An older version of MOCU is used for control of the USVs on the Mine Countermeasures Mission Module. Office of Naval Research (ONR) has also funded multiple efforts using MOCU as the display for modern game-like presentation layers and also the USV interface software for common architecture designs.

Figure 3. A display developed for an ONR effort to safely and efficiently control multiple USVs navigating a congested environment. The USV's video display is arranged to provide the operator with better spatial and situational awareness and a forward view from a second USV is in constant view. Most of the operator's interaction is by a standard gamepad, reducing training requirements for basic control substantially. All but the video and maps are HTML including the window borders, gauges, and text gauges. Parts of the display are many layers thick drawn by multiple modules, but display as one common interface.

MOCU is currently being updated to control the AeroVironment (AV) *Raven* and *Puma* on the *Tactical Robotic Controller (TRC)*. Under this effort, MOCU is demonstrated to control air and ground vehicles (PackBot and Talon) from one hardware form factor.

Large control systems require the software to be distributed to handle workload both at the operator and the hardware. SSC Pacific is currently developing a new control system based on web services built upon a three-tiered architecture focused on distribution but with the same design concepts used in MOCU. Remote Operator Control Services (ROCS) will use an HTML5 enabled browser to provide the presentation layer. *HTML5* and *WebGL* provide most of the capability provided within MOCU 3 but in a standardized format. Data is distributed by a data model built upon an Open Management Group (OMG) Data Distribution Service (DDS) standard from backend services.

8. SUMMARY

By necessity, any common OCU must be modular in order to support future robotic systems. However, modularity does not guarantee that an OCU can be expanded and grown to control robotic systems in the future. An OCU must support the ability to enhance or sometimes even wholly replace various modules in order to support new features, capabilities, or the user-interface widget *du jour*. Modularization of the underlying architecture also tends to cause the user interface to become 'modularized' as well, resulting in tiled displays that are busy, repetitive, and difficult to use. MOCU's publish/subscribe architecture allows modules to communicate with each other using a simple well-defined interface. This approach also isolates the modules, enabling them to be replaced without affecting any other modules, vastly simplifying future upgrades. MOCU avoids the tiled-interface pitfall by giving modules the ability to draw on the same screen area as other modules via layering and drawing-order constraints as well as the transparency capabilities inherent in *OpenGL*. The result is a modular, maintainable, expandable OCU that allows for the game-like displays needed by modern human-systems integration design.

REFERENCES

[1] D. Powell, G.A. Gilbreath, *Multi-robot Operator Control Unit (MOCU)*, SPIE Proc. 6230-67, 2006.
[2] G. Pardo-Castellote, *OMG Data-Distribution Service: Architectural Overview*, 23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03), 2003.